

第 17 章 Java 对数据库编程

本章介绍如何使用 Java 类库中的类来编写数据库应用程序。一般的数据库应用程序分为客户端和服务端两个部分。Java 的设计意图是将它用于服务器端，而将客户端交给其他语言编写的工具去处理。所以 Java 与 Delphi、VB 的一个重要区别是，它不提供可视化的数据库感应控件，因而编写客户端时如果采用 Java 作为开发工具会有比较多的编程工作。

17.1 Java 对数据库的连接

无论何种工具，要处理数据库必须做的第一件事情就是对数据进行连接。Java 提供了多种连接方式，这都是通过 JDBC 来进行的。本节将介绍 JDBC 的使用。

17.1.1 JDBC 的基本概念

JDBC 是 Java 数据库连接（Java Database Connectivity）的简写，是一组用于连接数据库以及执行 SQL 语句的 API。它允许用户从 JAVA 程序中访问任何支持 SQL 的关系型数据库，也允许用户访问其他的表格数据源，如 Excel 表格。

JDBC 最大的特点是无论后台是何种数据库，对于 Java 程序员而言，它的工作方式完全相同。JDBC 为许多不同的数据库连接模块的前端提供了统一的接口，这样就不用为连接不同的数据库而烦恼了。

17.1.2 JDBC 的工作方式

JDBC 功能强大，但使用简单。无论连接何种数据库，只要做好下列步骤即可：

（1）与数据源建立连接，包括数据库和电子表格。

通过 `DriverManager` 类建立与数据源的连接，这个连接将作为一个数据操作的起点，同时也是连接会话事务操作的基础。

（2）向数据库发送 SQL 命令。

通过 `Statement` 或者 `PreparedStatement` 类向数据源发送 SQL 命令。在命令发送后，调用类中相应的 `execute` 方法来执行 SQL 命令。

（3）处理数据源返回的结果。

数据库处理了 SQL 命令后，将返回处理结果。对于 DDL 和 DML 操作，将返回被修

改的记录数量。对于查询将返回一个 **ResultSet** 结果集，程序接着遍历这个结果集执行想要的操作就行了。

17.1.3 JDBC 驱动连接

要与数据源连接，需要所连数据源的驱动程序。JDBC 有四种连接方式：JDBC-ODBC 桥接方式、本地 API 部分 Java 驱动方式、JDBC-Net 纯 Java 驱动方式和本地协议纯 Java 驱动方式。下面分别介绍这些连接方式。

1. JDBC-ODBC桥加上ODBC驱动程序

ODBC 是 Windows 平台上使用最广泛的标准连接驱动。所有能在 Windows 上运行的数据库系统都提供了自己的 ODBC 驱动。通过这种方式，能访问所有的 Windows 上的数据库。但是它的效率稍低，而且本地机器上需要安装对应的 ODBC 驱动。下面演示了如何在本地机器上配置 ODBC 数据：

(1) 打开 ODBC 配置窗口。在控制面板中找到 ODBC 配置的图标。如果是 Windows XP，该图标在“管理工具”中，如果是 Windows 2000，该图标就在控制面板中。找到图标后，双击它，出现如图 17.1 所示的窗口。

(2) 添加用户数据源 (DSN)。单击图 17.1 中的“添加”按钮，出现如图 17.2 所示的窗口。在其中选择要添加的数据源的类型。这里选择的是 Access。

(3) 为数据源添加属性。这一步是为数据源添加一些属性，其中最重要的是数据源的名称，稍后 JDBC 要通过它来连接数据库文件。单击图 17.2 中的“完成”按钮，出现如图 17.3 所示的窗口。



图 17.1 ODBC 配置窗口

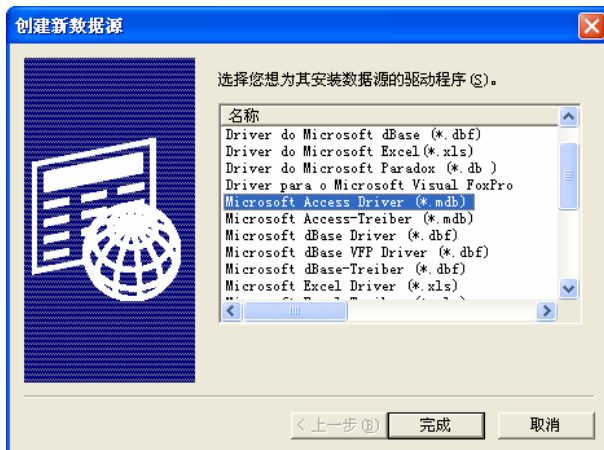


图 17.2 添加用户数据源

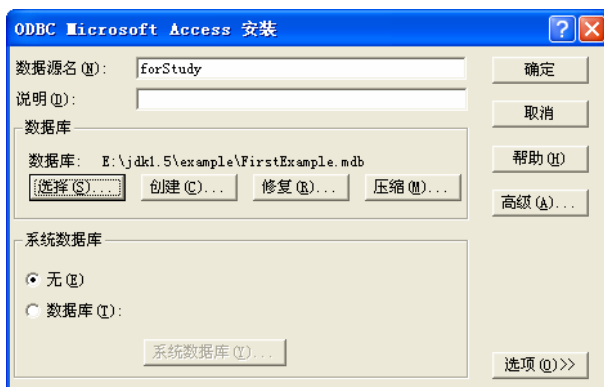


图 17.3 为数据源添加属性

这里将数据源的名称命名为“forStudy”，请记住该名称，后面编程要用到。然后单击“选择”按钮，会出现一个一般的文件选择对话框，在其中选择第 16 章创建的 FirstExample.mdb 文件即可。

完成后，单击“确定”按钮，回到图 17.1 所示界面上，会看到多出了一个用户数据源，名称为“forStudy”，表示配置成功。

2. 本地API部分Java驱动程序

这种连接方式如图 17.4 所示。

驱动程序使用本地 API 与数据源系统通信，使用 Java 方法调用执行数据操作的 API 函数。这就要求驱动程序要与应用程序一起驻留在客户层上，并直接与数据库服务器进行通信。因此，它要求客户机上有一些二进制代码。这种方法速度比 JDBC-ODBC 更快，但它必须使用本地代码，而且各个厂商提供的本地接口在驱动中不一致，编程时需要查找厂商提供的手册。

这种驱动连接方式最大的弱点是需要把开发商数据库加载到每一台客户机上，因此它不能应用于 Internet。而且驱动程序使用的是 Java Native Interface，由于该接口在 JVM 的不同开发商之间没有得到一致的实现，所以它在平台间的移植性能也不是很好。

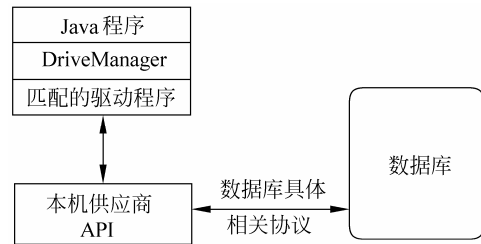


图 17.4 本地 API 连接方式

3. JDBC-Net纯Java驱动程序

这种连接方式的驱动程序采用一种三层化方法，JDBC 数据库请求凭借这种方法，被转换成数据库独立的网络协议，并转发给中间层服务器。然后中间层服务器再把该请求转换成数据库特定的本机连接接口，并把该请求传递给数据库服务器。如果中间层服务器是用 Java 编写成的，它可以使用前面介绍的两种类型的 JDBC 驱动程序来完成这些工作，这意味着它在体系结构上是非常灵活的。

总的体系结构由三层组成：JDBC 客户和驱动程序、中间件以及数据库。如图 17.5 所示：

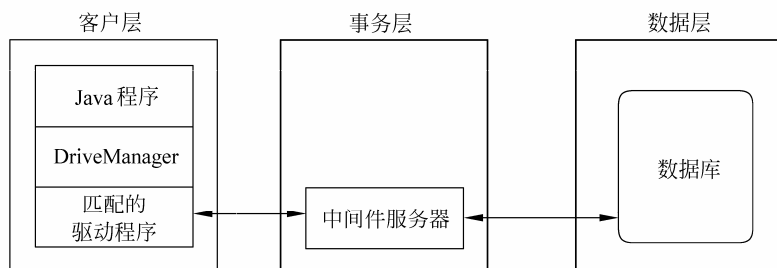


图 17.5 JDBC-Net 连接方式

在这种连接方式中，小 JDBC 驱动程序（通常只有几百 KB）在客户机上执行，并通过网络把 SQL 命令传递给 JDBC 服务器，然后接收来自服务器的数据，并管理连接。这种方式考虑到了在 Internet 上的部署。这种方式的优点是显而易见的：

- 在中间层上有一个组件，所以无需客户机上存在任何数据库。

- ❑ 它的可移植性能和查询性能都非常好，可以供多个用户并发操作数据库。
- ❑ 客户端的 JDBC 驱动程序非常小，加载迅速，适合在 Internet 上部署。
- ❑ 它为高速缓存之类的功能、加载平衡以及日志和审计之类的共计系统管理提供了支持。
- ❑ 大多数的三层 Web 应用程序都涉及到安全、防火墙以及代理，而这类驱动程序一般都提供了这些功能。

当然，这种方式也有其缺点：

- ❑ 要求数据库特定的编码在中间层完成，这增加了中间层设计者的负担，整个开发周期比前面两种方式要长。
- ❑ 遍历查询结果集时要花费比较长的时间，这是因为这些数据要经历后台服务器。

4. 本地协议的纯Java驱动程序

它允许从 Java 客户端直接调用连接到数据库服务器。它是纯粹的 Java 程序，不需要对客户端进行配置，只需要注册相应的驱动程序名称即可，这全面体现了 Java 的跨平台性和安全性。不过，当后台数据库变成一个不同开发商产品时（尽管这种情形很少见，但不能完全排除），不能使用同一个 JDBC 驱动程序，需要替换该驱动程序。

这些驱动程序把 JDBC 直接转换成 DBMS 开发商提供的 Java 驱动程序。这些驱动程序多数只能从 DBMS 开发商那里才能得到，因为只有开发商才最了解它们自己的协议。整个连接方式如图 17.6 所示。

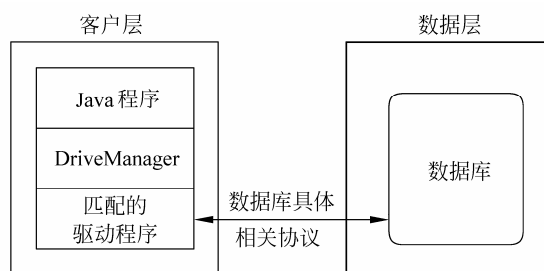


图 17.6 本地协议连接方式

这种方式的优点是：

- ❑ 它不必把数据库请求转换成 ODBC 或者传递给另外一个服务器，所以性能一般都很高，比第一和第二种方式都更好。
- ❑ 无需在客户端或者服务器端安装特殊软件，而且这些驱动程序可以被动态下载。

这种方式的缺点在于，程序员需要给每个数据库使用不同的 JDBC 驱动程序。

相对而言，在 Windows 平台上使用第一种方式较多，在 Linux/Unix 平台上使用第四种方式较多。

17.1.4 连接池

进行 JDBC 操作的第一步就是建立数据库连接。但这个过程是比较消耗资源的，如果

每进行一次数据库操作就建立一次连接，那么消耗在此动作上面的时间和资源，将大大影响程序的效率。

为解决这一问题，JDBC 使用了“连接池”的概念。在连接池中，保存了若干已经建立好的数据连接。每次需要与数据源通信的时候，如果所需要的连接在连接池里，那么直接使用这些现成的数据连接即可，使得运行速度大大提高。

数据库连接池运行在服务器端，而且对应用程序的编码没有任何影响。不过前提是，应用程序必须通过 `DataSource` 对象（一个实现 `javax.sql.DataSource` 接口的实例）的方式，代替原有通过 `DriverManager` 类来获得数据库连接的方式。一个实现 `javax.sql.DataSource` 接口的类可以支持也可以不支持数据库连接池，但是两者获得数据库连接的代码基本是相同的。

它的一般代码如下：

```
Context ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup("jdbc/openbase");
```

如果当前 `DataSource` 不支持数据库连接池，应用程序将获得一个和物理数据库连接对应的 `Connection` 对象。而如果当前的 `DataSource` 对象支持数据库连接池，应用程序将自动获得重用的数据库连接，而不用创建新的数据库连接。重用的数据库连接和新建立连接的数据库连接，使用上没有任何不同。应用程序可以通过重用的连接正常地连接数据库，进行访问数据的操作，完成操作后应显式地调用 `close()` 关闭数据库连接。

当关闭数据连接后，当前使用的数据库连接将不会被物理关闭，而是放回到数据库连接池中进行重用。不过对于数据库应用程序员而言，这个操作是透明的。

在 JDBC3.0 规范中，提供一个支持数据库连接池的框架。这个框架仅仅规定了如何支持连接池的实现，而连接池的具体实现，JDBC 3.0 规范并没有做相关的规定。通过这个框架。可以让不同角色的开发人员共同实现数据库连接池。

通过 JDBC3.0 规范可以知道，具体数据库连接池的实现可以分为 JDBC Driver 级和 Application Server 级。在 JDBC Driver 级的实现中，任何相关的工作均由特定数据库厂商的 JDBC Driver 的开发人员来具体实现，即 JDBC Driver 既需要提供对数据库连接池的支持，同时也必须对数据库连接池进行具体实现。而在 Application Server 级中数据库连接池的实现中，特定数据库厂商的 JDBC Driver 开发人员和 Application Server 开发人员，共同实现数据库连接池的实现（但是现在大多数 Application Server 厂商实现的连接池的机制和规范中提到有差异），其中特定数据库厂商的 JDBC Driver 提供数据库连接池的支持，而特定的 Application Server 厂商提供数据库连接池的具体实现。

这些具体的实现是比较高级的内容，限于本书的篇幅，不做深入的研究。

17.1.5 事务操作

有时候需要对多个数据表进行操作，只有对这几个表的操作都成功时，才能认为整个操作完成，这样的操作称为“事务操作”。如果某一个步骤失败，之前的各个操作都要取消，这种取消动作被称为“回滚（rollback）”。JDBC 中的事务操作是基于同一个数据连接的，各个连接之间互相独立。当数据连接断开后，一个事务就结束了。关于事务操作的方法，都位于接口 `java.sql.Connection` 中。

在 JDBC 中，事务操作默认是自动提交。也就是说，一条对数据库的更新表达式代表一项事务操作，操作成功后，系统将自动调用 `commit()` 来提交，否则将调用 `rollback()` 来回滚。同时，程序还可以通过调用 `setAutoCommit(false)` 来禁止自动提交。之后就可以把多个数据库操作的表达式作为一个事务，在操作完成后调用 `commit()` 来进行整体提交，倘若其中一个表达式操作失败，都不会执行到 `commit()`，并且将产生相应的异常。此时，就可以在异常捕获时调用 `rollback()` 进行回滚。这样做可以保持多次更新操作后，相关数据的一致性。下面是程序片段：

```
try {
    conn = DriverManager.getConnection("jdbc:odbc:forStudy","","")
    conn.setAutoCommit(false);           //禁止自动提交，设置回滚点
    stmt = conn.createStatement();
    stmt.executeUpdate("alter table ..."); //数据库更新操作 1
    stmt.executeUpdate("insert into table ..."); //数据库更新操作 2
    conn.commit();                       //事务提交
} catch (Exception ex) {
    ex.printStackTrace();
    try {
        conn.rollback();                 //操作不成功则回滚
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

读者现在不必完全弄懂这段代码的含义，等到看完了本节后面的内容，再回过来看这段代码，就会完全明白每条语句的作用。

JDBC API 支持事务对数据库的加锁，并且提供了 5 种操作支持，2 种加锁密度。5 种支持如下。

- ❑ `static int TRANSACTION_NONE`：禁止事务操作和加锁。
- ❑ `static int TRANSACTION_READ_UNCOMMITTED`：允许脏数据读写（dirty reads）、重复读写（repeatable reads）和影象读写（phantom reads）
- ❑ `static int TRANSACTION_READ_COMMITTED`：禁止脏数据读写、允许重复读写和影象读写。
- ❑ `static int TRANSACTION_REPEATABLE_READ`：禁止脏数据读写和重复读写，允许影象读写。
- ❑ `static int TRANSACTION_SERIALIZABLE`：禁止脏数据读写、重复读写和允许影象读写。

其中几个术语解释如下。

- ❑ 脏数据读写（dirty reads）：当一个事务修改了某一数据行的值而未提交时，另一事务读取了此行值。倘若前一事务发生了回滚，则后一事务将得到一个无效的值（脏数据）。


- ❑ 重复读写 (repeatable reads)：当一个事务在读取某一数据行时，另一事务同时在修改此数据行。则前一事务在重复读取此行时，将得到一个不一致的值。
- ❑ 影象读写 (phantomreads)：当一个事务在某一表中进行数据查询时，另一事务恰好插入了满足了查询条件的数据行。则前一事务在重复读取满足条件的值时，将得到一个额外的“影象”值。

JDBC 根据数据库提供的默认值来设置事务支持及其加锁，它有两种加锁密度：分别是表加锁和行加锁。在上述事务操作中，最后一项就是表加锁，而第 3、4 项则是行加锁。用程序设置加锁的方法如下：

```
Settransactionisolation(int level);
```

也可以用下面的方法来查看数据库的当前设置：

```
getTransactionIsolation();
```

 **注意：**某些数据库（如 oracle）中，数据库驱动对事务处理的默认值是 TRANSACTION_NONE，即不支持事务操作，所以需要在程序中手动进行设置。在进行手动设置时，数据库及其驱动程序必须要支持相应的事务操作才行。

上述设置随着加锁密度的增加，其事务的独立性随之增加，更能有效地防止事务操作之间的冲突。同时，增加了加锁的开销，降低了用户之间访问数据库的并发性，程序的运行效率也会随之降低。因此程序员必须平衡程序运行效率和数据一致性之间的冲突。一般来说，对于只涉及到数据库的查询操作时，可以采用 TRANSACTION_READ_UNCOMMITTED 方式；对于数据查询远多于更新的操作，可以采用 TRANSACTION_READ_COMMITTED 方式；对于更新操作较多的，可以采用 TRANSACTION_REPEATABLE_READ；在数据一致性要求更高的场合再考虑最后一项，由于涉及到表加锁，因此会对程序运行效率产生较大的影响。

17.2 Java 对数据库的操作

连接上数据库之后，就可以对数据库中的数据进行操作了。无论何种语言、何种数据库，也无论多么复杂的系统，对数据库的操作都不外乎 5 种基本操作：增、删、改、查找和排序。在 Java 中，这些操作都是通过 SQL 语言来实现的。

17.2.1 常用接口及类

要操作数据库，多数需要通过下面的类以及接口来实现（本节的后面将通过一些例子来介绍这些类和接口的使用）。

(1) DriverManager 类

DriverManager 类是 JDBC 的管理层，作用于用户和驱动程序之间。它跟踪可用的驱动

程序，并在数据库和相应驱动程序之间建立连接。另外，`DriverManager` 类也处理诸如驱动程序登录时间限制，及登录和跟踪消息的显示等事务。

对于简单的应用程序，一般程序员需要在此类中直接使用的唯一方法是：

```
DriverManager.getConnection()
```

正如名称所示，该方法将建立与数据库的连接。`JDBC` 允许用户调用 `DriverManager` 的方法 `getDriver()`、`getDrivers()` 和 `registerDriver()`。但多数情况下，`DriverManager` 类自己管理建立连接的细节为上策。

(2) `Connection` 接口

该类对应于数据库连接对象，是 `JDBC` 操作的起点，同时也是一个 `JDBC` 事务的起点，封装了对数据库连接的操作。一般情况下，`Connection` 对象是由 `DriverManager.getConnection()` 方法来得到的，程序只需要获取该方法返回的一个指向 `Connection` 对象的引用，就可以对数据库进行操作。`Connection` 的方法比较多，但多数情况下，程序员只需要用到下面的两个方法。

- ❑ `Statement createStatement()`：创建一个 `Statement` 对象，并返回该对象将用于执行具体的 `SQL` 命令。
- ❑ `void close()`：关闭数据库连接，释放资源。

(3) `Statement` 接口

这是执行 `SQL` 命令的主要容器，它一次只能执行一条 `SQL` 命令。它通过三个不同的方法来执行 `SQL` 命令：

- ❑ `ResultSet executeQuery (String sql)`：执行 `sql` 命令，返回一个结果集合。通常用于执行 `SELECT` 命令。
- ❑ `int executeUpdate (String sql)`：执行 `sql` 命令，返回操作成功的记录条数。通常用于执行 `INSERT`、`UPDATE` 或 `DELETE` 命令。
- ❑ `boolean execute (String sql)`：执行 `sql` 命令，返回执行结果的标志。如果值为 `true`，表示返回了一个结果集，需要用 `getResultSet()` 方法获取这个结果集，也可以使用 `getMoreResults()` 获取子结果集。如果为 `false`，表示没有结果集，只需要调用 `getUpdateCount()` 方法获取记录更新的条数。

程序员需要根据执行的命令来选择合适的执行方法。

(4) `PreparedStatement` 接口

与 `Statement` 类似，但它对 `SQL` 命令进行预编译，对于需要多次执行的 `SQL` 语句而言，可以提高执行效率。作为 `Statement` 的子接口，它不仅拥有 `Statement` 的所有方法，还增加了一套方法，通过使用一个称为“占位符”的输入参数，程序员可以方便地将程序中的变量转换成为 `SQL` 语句中的变量，并降低出错的可能性。

(5) `ResultSet` 接口

它用来接受 `SQL` 语句执行后的结果，程序员可以通过 `next()`、`previous()` 等方法来访问结果集中的任意一条记录。同时它还提供了大量辅助方法，方便访问记录中的各个字段。关于这个接口，将在 17.2.7 中详细介绍。

17.2.2 建立数据库连接

前面介绍过,建立数据库连接有 4 种方式,这里以 Windows 系统下常用的 JDBC-ODBC 连接为例来讲解。

为连接到某个数据库,首先要建立一个 JDBC-ODBC 桥接器,它将 JDBC 操作转换成 ODBC 操作来完成。这需要使用 sun.jdbc.odbc 包和 java.lang 包中的类来实现。

(1) 加载 ODBC 驱动,使用下面的方法:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

该方法是一个静态方法,可以直接使用。同时,它还可能抛出异常,所以需要写在 try-catch 语句块中。注意,这里是加载 ODBC 驱动,无论后台是何种数据库,该驱动程序是不变的。如果不是采用 JDBC-ODBC 桥,而采用其他的 Java 驱动,就需要加载对应的驱动程序。例如,后台是 MySQL,加载驱动程序的代码是:

```
Class.forName("org.glt.mm.mysql.Driver");
```

如果是 DB2,则加载方式为:

```
Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
```

其余的数据库,请查阅数据库帮助。

(2) 建立连接


加载了驱动程序后,就可以建立连接了。这需要使用 java.sql 中的 Connection 类来声明一个对象,再使用类 DriverManager 的静态方法 getConnection 创建连接对象。该方法的声明如下:

```
static Connection getConnection(String url, String user, String password);
```

第一个参数是指定要连接的数据库名称,该数据库既可以是本地的,也可以是远程的。后两个参数分别是用户名和密码,如果数据库没有该项设置,可以为空。

如果使用 ODBC 驱动,则要写成这个样子:

```
jdbc:odbc:数据源名字
```

 **注意:** 这里是数据源而非数据库名称。

如果是非 ODBC 驱动,例如 MySQL,则要写成:“jdbc:mysql://数据库主机名/数据库名称”。

【例 17.1】 使用 JDBC-ODBC 桥来连接一个 Access 数据库。

该数据库的名称为 FirstExample,在 ODBC 数据源中的名称为“forStudy”,用户和口令均为空。这就是第 16 章建立的数据库,以及 17.1 节配置的 ODBC 数据源。

//-----连接数据库示例程序段,程序编号 17.1-----

```
try{
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");//加载数据库驱动
```

```

        Connection con=DriverManager.getConnection
            ("jdbc:odbc:forStudy","",""); //这里是 ODBC 数据源名称
    }catch(ClassNotFoundException e){
    }catch(SQLException e){
    }
}

```

17.2.3 关闭数据库连接

调用 Connection 的 close()方法可以关闭数据库连接。但在实际应用中,可能会出现各种异常情况,程序就不会按照正常流程运行,就可能出现 close()方法没有被调用的情况,而程序即便终止了运行,仍然会占用数据库的连接数量,这就是“连接泄漏”。当数据库的连接数量达到极限后,就会导致数据库不再响应用户请求,甚至停止等后果,这是必须要避免的。所以,一般将 close 写在 try-catch-finally 的 finally 语句中,保证即便发生异常,数据库也一定会关闭。

【例 17.2】 关闭数据库连接示例。

//-----关闭数据库示例程序段, 程序编号 17.2-----

```

try{
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    Connection con=DriverManager.getConnection
        ("jdbc:odbc:forStudy","","");
}catch(ClassNotFoundException e){
}catch(SQLException e){
}finally{ //在这里关闭数据库
    if(con!=null){
        try{
            if(!con.isClosed())
                con.close(); //关闭数据库连接
        }catch(SQLException el){
            el.printStackTrace();
        } //end try-catch
    } //end if
} //end try-catch-finally

```

17.2.4 添加新数据

获得连接后,程序就可以操作数据库了,先来看如何添加数据。这需要使用 SQL 语句,一般将 SQL 语句存放在一个 Statement 对象中,然后执行该对象的 executeUpdate()方法就可以了。

【例 17.3】 向数据库中添加新数据。

例如,在数据库 FirstExample 中有一张表名为 student,它具有 code、name、sex、address、room、tel 6 个字段(如果您的数据库中该表的字段不是这个样子,请将其设置成这样 6 个字段),希望添加一条记录:(30, 小王, 男, 湖南湘潭, N-408, 8293456), 程序代码如下:

//-----程序名 insertData.java, 程序编号 17.3-----

```
//本程序测试插入数据
import java.sql.*;
public class insertData {
    public static void main(String[] args) {
        Connection con=null;
        Statement stmt;
        //要执行的 SQL 语句
        String sqlString="insert into student values('30', '小王', '男',
                                                    '湖南湘潭', 'N-408', '8293456')";

        try{
            //加载数据库驱动
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            //连接数据库
            con=DriverManager.getConnection("jdbc:odbc:forStudy","","");
            stmt=con.createStatement();
            //执行插入命令
            stmt.executeUpdate(sqlString);
            System.out.println("插入成功");
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            try{
                con.close();
            }catch(SQLException e){
                e.printStackTrace();
            }
        }
    }
}
```

执行该程序后，可用 Access 打开 student 表来查看插入数据是否成功。

上面的程序实现比较简单，但没有多大的实际意义，因为在实际运行中，需要插入的数据通常都是由用户输入的，是一些变量。而将变量组配成一条 SQL 语句是一件很容易出错的事。例如有变量：code="30"，name="小王"，sex="男"，address="湖南湘潭"，room="N-408"，tel="8293456"那么像这样组配：

```
sqlString="insert into student values('code','name','sex','address',
'room','tel')";
```

是肯定错误的，被代入到 SQL 语句中去的是“code”、“name”这样的变量名，而非它所存储的字符串。所以应该写成这个样子：

```
sqlString="insert into student values('"+code+"','"+name+"','"+sex+"',
'"+address+"','"+ room + "','"+ tel + "',')";
```

这样写，才会将这些变量本身存储的值代入到 SQL 语句中。一个可以接受用户输入的程序，如下所示：

//-----程序名 insertData.java, 程序编号 17.4-----

```

import java.sql.*;
import java.io.*;
public class insertData {
    public static void main(String[] args) {
        Connection con=null;
        Statement stmt;
        String sqlString;
        String name,sex,address,code,room,tel;
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            con=DriverManager.getConnection("jdbc:odbc:forStudy","","");
            stmt=con.createStatement();
            code=getInput("请输入编号: ");
            name=getInput("请输入姓名: ");
            sex=getInput("请输入性别: ");
            address=getInput("请输入地址: ");
            room=getInput("请输入寝室: ");
            tel=getInput("请输入电话: ");
            //拼装 SQL 字符串
            sqlString="insert into student values('"+code+"','"+name+"','"+
                + sex + "','" + address + "','" + room + "','" + tel
                + "')";
            stmt.executeUpdate(sqlString);
            System.out.println("插入成功");
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            try{
                con.close();
            }catch(SQLException e){
                e.printStackTrace();
            }
        }
    }
    //接收用户输入
    public static String getInput(String msg){
        String result=null;
        try{
            //创建用户输入流
            BufferedReader in=new BufferedReader(new InputStreamReader
                (System.in) );
            System.out.print(msg);
            result=in.readLine();
        }catch(IOException e){
            e.printStackTrace();
        }
        return result;
    }
}

```

程序运行截图如图 17.7 所示:

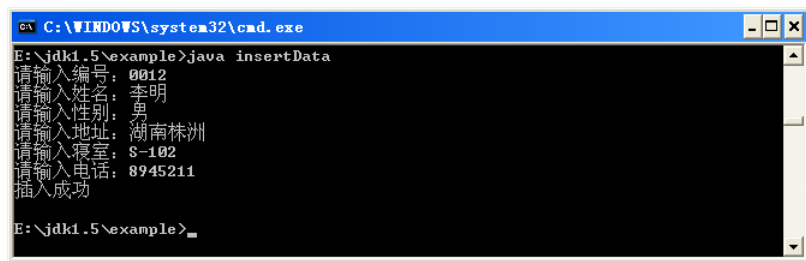


图 17.7 程序运行截图

从程序 17.4 中可以看出，将程序中的变量还原成 SQL 命令可以使用的字符串是件很容易出错的工作，如果字符串本身中还含有双引号或单引号，转换就会变得更为困难。即便是非常熟练的程序员，也很可能在这里犯错。所以，Java 又提供了一个 `PreparedStatement` 类，该类一个重要的作用就是提供了一个“占位符”，可以方便程序员进行转换工作。上面的例子可以改成：

```
sqlString="insert into student values(?,?,?,?)"; //?就是占位符
```

然后用下面的程序：

```
PreparedStatement ps=con.prepareStatement(sqlString);
ps.setString(1,code);      //替换第一个占位符
.....
ps.setString(4,address);   //替换第四个占位符
ps.executeUpdate();
```

注意上面的第一个占位符的设置。在 Access 中，数值型字段其实仍然是以字符串来存储的，所以即便 `code` 是整型数据，仍然可以使用 `setString` 这种方式来为数值型字段赋值。而对于其他的大型数据库，是严格区分 `int`、`long`、`short` 等类型的，就必须要用 `setInt` 或 `setLong` 等来赋值。整个程序如程序 17.5 所示：

//-----程序名 insertData.java，程序编号 17.5-----

```
import java.sql.*;
import java.io.*;
public class insertData {
    public static void main(String[] args) {
        Connection con=null;
        PreparedStatement ps;
        String sqlString;
        String name,sex,address,code,room,tel;
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            con=DriverManager.getConnection("jdbc:odbc:forStudy","","");
            code=getInput("请输入编号: ");
            name=getInput("请输入姓名: ");
            sex=getInput("请输入性别: ");
            address=getInput("请输入地址: ");
            room=getInput("请输入寝室: ");
            tel=getInput("请输入电话: ");
```

```

        sqlString="insert into student values(?,?,?,?,?,?)";
        ps=con.prepareStatement(sqlString);
        ps.setString(1,code); //替换第一个占位符
        ps.setString(2,name);
        ps.setString(3,sex);
        ps.setString(4,address);
        ps.setString(5,room);
        ps.setString(6,tel);
        ps.executeUpdate(); //执行 SQL 命令
        System.out.println("插入成功");
    }catch(Exception e){
        e.printStackTrace();
    }finally{
        try{
            con.close();
        }catch(SQLException e){
            e.printStackTrace();
        }
    }
}

public static String getInput(String msg){
    String result=null;
    try{
        BufferedReader in=new BufferedReader(new InputStreamReader
            (System.in));
        System.out.print(msg);
        result=in.readLine();
    }catch(IOException e){
        e.printStackTrace();
    }
    return result;
}
}

```

这个程序的运行情况和程序 17.4 是完全一样的。

17.2.5 删除数据

删除数据很简单，只要执行 SQL 语句中的删除命令 **delete** 即可。**delete** 命令本身比较简单，只是仍然需要 Java 中字符串的组配与 SQL 命令字符串之间的差异。下面是一个简单的例子。

【例 17.4】 从数据库中删除记录。

//-----程序名 deleteData.java, 程序编号 17.6-----

```

import java.sql.*;
import java.io.*;
public class deleteData {
    public static void main(String[] args) {
        Connection con=null;

```

```

Statement stmt;
String sqlString;
String code;
int k;
try{
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    con=DriverManager.getConnection("jdbc:odbc:forStudy","","");
    code=getInput("请输入要删除记录的编号: ");
    sqlString="delete from student where code='"+code+"'";
    stmt=con.createStatement();
    //执行删除命令, 并获取成功执行的记录数
    k=stmt.executeUpdate(sqlString);
    System.out.println("删除了"+k+"条记录");
}catch(Exception e){
    e.printStackTrace();
}finally{
    try{
        con.close();
    }catch(SQLException e){
        e.printStackTrace();
    }
}
}

public static String getInput(String msg){
    String result=null;
    try{
        BufferedReader in=new BufferedReader(new InputStreamReader
            (System.in));
        System.out.print(msg);
        result=in.readLine();
    }catch(IOException e){
        e.printStackTrace();
    }
    return result;
}
}

```

17.2.6 修改数据

使用 SQL 语句中 `update` 命令就可以修改数据, 该命令第 16 章已经介绍过。下面是个简单的例子, 将所有性别为“男”的记录改成“女”。

【例 17.5】 修改数据示例。

//-----程序名 updateData.java, 程序编号 17.7-----

```

import java.sql.*;
import java.io.*;
public class updateData {
    public static void main(String[] args) {
        Connection con=null;

```

```

Statement stmt;
String sqlString;
int k;
try{
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    con=DriverManager.getConnection("jdbc:odbc:forStudy","","");
    //SQL 命令字符串
    sqlString="update student set sex='女' where sex='男'";
    stmt=con.createStatement();
    //执行 SQL 命令
    k=stmt.executeUpdate(sqlString);
    System.out.println("修改了"+k+"条记录");
}catch(Exception e){
    e.printStackTrace();
}finally{
    try{
        con.close();
    }catch(SQLException e){
        e.printStackTrace();
    }
}
}
}

```

17.2.7 查询数据

利用 SELECT 语句查询得到结果后，会将结果放在一个 **ResultSet** 集中，遍历这个集合就可以做任何需要的操作。**ResultSet** 是由 Java 提供的一个非常有用的类，每当查询语句执行完成后，就会返回一个查询结果的集合给它。可以将它想像成一张表，只包含那些查询符合条件的行和列。这张表有一个游标（cursors），可以把它想像成一根指针，它始终指向记录集中的某一行，所有对记录集的操作，默认都是对这根游标所指向的记录的操作。因此 **ResultSet** 提供了大量的方法供程序员来管理这根游标。

ResultSet 的方法非常多，表 17.1 列出了其中一些常用的方法。

表 17.1 **ResultSet**的常用方法

方 法	说 明
boolean absolute(int row)	将游标移动到指定行位置（绝对值）
void afterLast()	将游标移动到结果集的最后一记录的后面
void beforeFirst()	将游标移动到结果集的第一条记录的前面
void cancelRowUpdates()	取消本行数据的更新
void clearWarnings()	清除本对象的所有警告
void close()	立即关闭与结果集连接的数据库并释放JDBC资源
void deleteRow()	从结果集以及数据表中删除游标所指向的记录
int findColumn(String columnName)	查找结果集中是否有给定的字段名称
boolean first()	将游标移动到结果集的第一条记录处

续表

方 法	说 明
BigDecimal getBigDecimal(int columnIndex)	将游标指向的记录中由columnIndex所确定的字段转换成BigDecimal数据返回
boolean getBoolean(int columnIndex)	将游标指向的记录中由columnIndex所确定的字段转换成boolean数据返回
byte getByte(int columnIndex)	将游标指向的记录中由columnIndex所确定的字段转换成byte数据返回
String getCursorName()	获取当前使用的游标名称
Date getDate(int columnIndex)	将游标指向的记录中由columnIndex所确定的字段转换成Date数据返回
double getDouble(int columnIndex)	将游标指向的记录中由columnIndex所确定的字段转换成double数据返回
int getInt(int columnIndex)	将游标指向的记录中由columnIndex所确定的字段转换成int数据返回
long getLong(int columnIndex)	将游标指向的记录中由columnIndex所确定的字段转换成long数据返回
int getRow()	获取游标所指向的行数
short getShort(String columnName)	将游标指向的记录中由columnIndex所确定的字段转换成short数据返回
Statement getStatement()	返回与结果集相关联的Statement对象
String getString(int columnIndex)	将游标指向的记录中由columnIndex所确定的字段转换成String数据返回
int getType()	返回结果集对象的类型，它的类型是下列三个之一： ResultSet.TYPE_FORWARD_ONLY; ResultSet.TYPE_SCROLL_INSENSITIVE; ResultSet.TYPE_SCROLL_SENSITIVE
void insertRow()	将准备插入的记录插入到结果集以及对应的表中
boolean isAfterLast()	测试游标是否在最后一条记录的后面
boolean isBeforeFirst()	测试游标是否在第一条记录的前面
boolean isFirst()	测试游标是否在第一条记录处
boolean isLast()	测试游标是否在最后一条记录处
boolean last()	将游标移动到最后一条记录处
void moveToCurrentRow()	将游标移动到前面记录下来的位置处，通常是当前位置。如果处于插入状态，此方法无效
void moveToInsertRow()	将游标移动到要插入记录的位置
boolean next()	将游标向后移动一行
boolean previous()	将游标向前移动一行
void refreshRow()	刷新游标所指向的记录值
boolean relative(int rows)	将游标移动到相对于当前位置差rows行的位置
boolean rowDeleted()	测试行是否已被删除
boolean rowInserted()	测试行是否已被插入
boolean rowUpdated()	测试行是否已被更新
void updateByte(int columnIndex, byte x)	将当前记录的columnIndex字段用x的值来取代

续表


方 法	说 明
void updateDate(int columnIndex, Date x)	将当前记录的columnIndex字段用x的值来取代
void updateDouble(int columnIndex, double x)	将当前记录的columnIndex字段用x的值来取代
void updateInt(int columnIndex, int x)	将当前记录的columnIndex字段用x的值来取代
void updateLong(int columnIndex, long x)	将当前记录的columnIndex字段用x的值来取代
void updateRow()	更新与结果集相联系的表中的当前行
void updateShort(String columnName, short x)	将当前记录的columnIndex字段用x的值来取代
boolean wasNull()	测试读入进来的最后一列是否为SQL NULL值

下面的例子中先用 SQL 查询获取一个结果集，然后遍历这个结果集，将其中的记录输出到屏幕。

【例 17.6】 查询数据示例。

//-----程序名 queryData.java, 程序编号 17.8-----

```
import java.sql.*;
import java.io.*;
public class queryData{
    public static void main(String[] args) {
        Connection con=null;
        Statement stmt;
        ResultSet rs;
        String name,sex,address,code;
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            con=DriverManager.getConnection("jdbc:odbc:forStudy","","");
            stmt=con.createStatement();
            //执行查询命令，并获取返回的结果集
            rs=stmt.executeQuery("select * from student");
            //下面开始遍历结果集
            while(rs.next()){ //游标向后移动
                name=rs.getString("name"); //获取 name 字段的内容
                sex=rs.getString("sex"); //获取 sex 字段的内容
                address=rs.getString("address");
                code=rs.getString("code");
                System.out.println(name+" "+sex+" "+address+" "+code);
            }
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            try{
                con.close();
            }catch(SQLException e){
                e.printStackTrace();
            }
        }
    } //try end
} //method end
} //class end
```

 **注意：**在程序 17.8 中，需要先将游标执行一次 `next()` 方法，才能读取数据。这是因为游标一开始的时候位于第一条记录的前面。

17.3 学生信息管理系统实例

本节介绍一个用 Java 编制的小小的学生信息管理系统，这是一个 GUI 界面的数据库应用系统。前面介绍过，Java 没有像 Delphi 那样提供可视化的数据库感应控件，所以编制这类程序时，程序员必须花费较多的精力在处理可视化组件对数据的显示上面。而同样是对数据库进行操作，JSP 程序写起来就要简单得多，因为它只需要处理数据库的读写部分，显示部分交给前台的浏览器处理。

在开始设计程序之前，先看看程序第一次运行时界面：



图 17.8 程序第一次运行时截图

这里用到的数据库仍然是前面的 FirstExample，ODBC 数据源名称为 forStudy，表名是 student，拥有 6 个字段，如表 17.2 所示：


表 17.2 student 表中各个字段说明

字段名	类型	长度	含义	主键否
code	文本	10	学生代码（学号）	是
name	文本	10	名称	否
sex	文本	2	性别	否
address	文本	20	家庭住址	否
room	文本	20	寝室	否
tel	文本	20	电话	否

17.3.1 程序设计思路

程序的基本思路是，利用 SQL 语句将所有数据全部读入到 ResultSet 中，然后利用该

对象中的各种方法（包括添加、删除等）对数据进行处理。

 **注意：**某些数据库不支持数据集的这些功能，只能用相应的 SQL 命令来实现。

一般情况下，这种 MIS 程序有三种编程思路：

- ❑ 浏览页面和编辑页面分别是不同的页面。这样编程比较简单，但使用者不方便。
- ❑ 浏览页面和编辑页面是同一个页面，而且浏览的同时随时可以编辑，编辑后也不需要用户保存，只要离开本条记录，就自动保存。这样实现使用者最方便，但编程最为困难。
- ❑ 浏览页面和编辑页面是同一个页面，但浏览时不能编辑，编辑了记录后需要用户按下按钮保存。通过按钮在两种模式间切换。编程的难度和使用的方便程度位于上述两者之间。

本程序采用的是第三种思路。

17.3.2 几个相关标记

在此程序中，需要前后移动游标浏览数据，还可以切换到编辑模式修改或增加记录，也可以删除记录。这需要精确记录游标的位置和记录条数。**ResultSet** 对这两项的支持都很弱，直接编程不太方便，需要自己增加几个类成员作为相关标记：

```
protected int recordState=onlyRead;           //当前记录的状态
public static final int onlyRead=0, adding=1,amending=2; //记录的三种状态
protected int curRow=0,                       //游标的位置
               recordCnt=0;                   //记录总数
```

ResultSet 不能直接获取记录总数，需要使用 SQL 语句：

```
select count(*) from student
```

来实现，然后用再设置 **curRow** 和 **recordCnt** 的值。

17.3.3 程序界面设计

首先来设计程序的界面，这个比较简单，用第 14 章介绍的知识就可以轻松完成，只有一点需要注意。因为在退出程序时必须关闭数据库，所以需要监听窗口的 **windowClosing** 事件。另外，某些按钮在一开始应该处于不可用状态，例如，“前一条”、“后一条”等，所以需要设置一下。

程序界面设计的程序代码如下：

//-----程序名 AddressList.java，程序编号 17.9-----

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.sql.*;
public class AddressList extends WindowAdapter{
```

```

JFrame    mainJframe;
Container con;
JPanel    pane[];
JTextField fieldText[];
JLabel    lbl[];
JButton    firstBtn,preBtn,nextBtn,lastBtn,
           addBtn,editBtn,delBtn,cancelBtn,saveBtn;
Connection conn=null;
Statement  stmt=null;
ResultSet  rs=null;
protected int recordState=onlyRead, curRow=0,recordCnt=0;
public static final int onlyRead=0, adding=1,amending=2;
public static final String lblmsg[]={"学号","姓名","性别","家庭住址",
  "寝室","电话"};
private static final int fieldCnt=6;
//在构造方法中布置界面
public AddressList() {
    mainJframe = new JFrame("学生信息管理");
    con=mainJframe.getContentPane();
    con.setLayout(new BoxLayout(con,BoxLayout.Y_AXIS));
    pane=new JPanel[4];
    for(int i=0; i<4; i++){
        pane[i]=new JPanel();
        pane[i].setLayout(new FlowLayout());
    }
    fieldText = new JTextField[fieldCnt];
    lbl = new JLabel[fieldCnt];
    for(int i=0; i<fieldCnt; i++){
        fieldText[i]=new JTextField();
        fieldText[i].setColumns(10);
        fieldText[i].setEditable(false);
        lbl[i]=new JLabel();
        lbl[i].setText(lblmsg[i]);
        pane[i/2].add(lbl[i]);
        pane[i/2].add(fieldText[i]);
    }
    fieldText[2].setColumns(6);
    fieldText[3].setColumns(12);
    firstBtn=new JButton("第一条");
    preBtn=new JButton("前一条");
    nextBtn=new JButton("后一条");
    lastBtn=new JButton("最后一条");
    addBtn=new JButton("增加记录");
    editBtn=new JButton("编辑记录");
    delBtn=new JButton("删除记录");
    cancelBtn=new JButton("取消改变");
    saveBtn=new JButton("保存记录");
    firstBtn.addActionListener(this);
    preBtn.addActionListener(this);
    nextBtn.addActionListener(this);
    lastBtn.addActionListener(this);
}

```

```

        addBtn.addActionListener(this);
        editBtn.addActionListener(this);
        delBtn.addActionListener(this);
        cancelBtn.addActionListener(this);
        saveBtn.addActionListener(this);
        pane[3].add(firstBtn);
        pane[3].add(preBtn);
        pane[3].add(nextBtn);
        pane[3].add(lastBtn);
        pane[3].add(addBtn);
        pane[3].add(editBtn);
        pane[3].add(delBtn);
        pane[3].add(cancelBtn);
        pane[3].add(saveBtn);
        for(int i=0;i<4;i++)
            con.add(pane[i]);
        mainJframe.setSize(450,300);
        mainJframe.setVisible(true);
        mainJframe.addWindowListener(this);
        connection();
        if (recordCnt>0) showDate();
        setFace();
    }
    //设置按钮的初始状态
    protected void setFace(){
        firstBtn.setEnabled(false);
        preBtn.setEnabled(false);
        nextBtn.setEnabled(true);
        lastBtn.setEnabled(true);
        addBtn.setEnabled(true);
        editBtn.setEnabled(true);
        delBtn.setEnabled(true);
        cancelBtn.setEnabled(false);
        saveBtn.setEnabled(false);
    }
    public static void main(String[] args) {
        new AddressList();
    }
}

```


其中的 `connection()`和 `showDate()`方法将在后面介绍。这里实际上已经提供了本程序的框架，后面的代码都要添加在 `AddressList.java` 文件中。

17.3.4 打开数据库

程序要做的第一件事情就是连接数据库，这里使用的方法前面已经介绍过。但默认情况下，数据集中的数据是只可读，游标也只能向后移动，与程序的要求不符。所以要使用下面的方法：

```
conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,ResultSet.
CONCUR_UPDATABLE);
```

第一个参数指定游标可以前后移动，第二个参数指定数据集可读写。

 **注意：**某些数据库不支持这些功能。

在打开数据库的同时，需要做两件事情：一是获取数据，二是统计记录的条数。这要用两条 SQL 语句来完成。打开数据库写成一个方法，代码如下：

```
public void connection(){
    try{
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        conn=DriverManager.getConnection("jdbc:odbc:forStudy","","");
        stmt=conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                    ResultSet.CONCUR_UPDATABLE);
        rs=stmt.executeQuery("select count(*) from student");
        if(rs.next())
            recordCnt=rs.getInt(1); //获取记录总数
        rs=stmt.executeQuery("select code,name,sex,address,room,
        tel from student");
        rs.next(); //将游标移动到第一条记录处
        curRow=1;
    }catch(SQLException e){
        JOptionPane.showMessageDialog(mainJframe,"数据库无法连接或没有记录");
    }catch(ClassNotFoundException e){
        JOptionPane.showMessageDialog(mainJframe,"无法加载 ODBC 驱动");
    }
}
```

17.3.5 关闭数据库

程序在使用中是不需要关闭数据库的，只有当程序退出运行时才需要关闭数据库，所以需要将关闭数据库的代码写在 windowClosing 事件的响应代码中。程序如下：

```
//退出系统时要关闭数据库
public void windowClosing(WindowEvent e1){
    try{
        conn.close();
    }catch(SQLException e){
        e.printStackTrace();
    }finally{
        System.exit(0);
    }
}
```

17.3.6 显示数据到控件中

在大多数情况下，界面上显示的记录必须是游标指向的记录，而默认情况下，界面上

的组件是不知道游标移动情况的，也无法自动显示记录中的数据。所以需要有一个方法来将当前记录显示在组件中，一旦游标移动，程序就要调用此方法。

```
//依次在 text 中显示"学号","姓名","性别","家庭住址","寝室","电话"
public void showDate(){
    try{
        fieldText[0].setText(rs.getString("code"));
        fieldText[1].setText(rs.getString("name"));
        fieldText[2].setText(rs.getString("sex"));
        fieldText[3].setText(rs.getString("address"));
        fieldText[4].setText(rs.getString("room"));
        fieldText[5].setText(rs.getString("tel"));
    }catch(SQLException e){
        JOptionPane.showMessageDialog(mainJframe,"无法获取数据");
    }
}
```

17.3.7 几个辅助方法

为了编程上的方便，需要将几个按钮都要用到的功能写成下面两个方法供其他方法调用：

```
//设置文本框的读写状态
protected void setTextState(boolean flag){
    for(int i=0;i<fieldCnt;i++){
        fieldText[i].setEditable(flag);
    }
}
//将所有文本框中的数据清除
protected void setTextEmpty(){
    for(int i=0;i<fieldCnt;i++){
        fieldText[i].setText(null);
    }
}
```

17.3.8 “第一条”按钮事件响应代码

当用户单击“第一条”按钮时，需要做下面几件事情：

- ☐ 将游标移动到第一条记录处；
- ☐ 显示记录到界面上；
- ☐ 修改按钮的使用状态；
- ☐ 修改相应的变量值。

```
protected void doMoveFirst(){
    //容错处理
    if (curRow<=1){
        firstBtn.setEnabled(false);
        preBtn.setEnabled(false);
        curRow=1;
    }
}
```



```

        return ;
    }
    try{
        if (rs.first()){ //移动游标到第一条位置
            showDate(); //显示当前记录到界面上
            curRow=1; //记录游标的新位置
            //重新设置按钮的状态
            firstBtn.setEnabled(false);
            preBtn.setEnabled(false);
            nextBtn.setEnabled(true);
            lastBtn.setEnabled(true);
        }
    }catch(SQLException el){
        JOptionPane.showMessageDialog(mainJframe,"移动游标出错");
    }
}

```

17.3.9 “前一条”按钮事件响应代码

当用户单击“前一条”按钮时，需要做下面几件事情：

- ☐ 将游标移动到前一条记录处；
- ☐ 显示记录到界面上；
- ☐ 判断当前游标所在位置，修改按钮的使用状态；
- ☐ 修改相应的变量值。

```

protected void doMovePrevior(){
    if (curRow<=1){ //容错处理
        firstBtn.setEnabled(false);
        preBtn.setEnabled(false);
        curRow=1;
        return ;
    }
    try{
        if (rs.previous()){ //向前移动游标
            showDate();
            curRow--;
            if (curRow==1){ //如果是第一条记录
                firstBtn.setEnabled(false);
                preBtn.setEnabled(false);
            }
            nextBtn.setEnabled(true);
            lastBtn.setEnabled(true);
        }
    }catch(SQLException el){
        JOptionPane.showMessageDialog(mainJframe,"移动游标出错");
    }
}

```

17.3.10 “后一条”按钮事件响应代码

当用户单击“后一条”按钮时，它执行的操作恰好和“前一条”相反，不再赘述。

```
protected void doMoveNext(){
    if (curRow>=recordCnt){           //容错处理
        nextBtn.setEnabled(false);
        lastBtn.setEnabled(false);
        curRow=recordCnt;
        return ;
    }
    try{
        if (rs.next()){               //向后移动游标
            showDate();
            curRow++;
            if (curRow==recordCnt){ //如果是最后一条记录
                nextBtn.setEnabled(false);
                lastBtn.setEnabled(false);
            }
            firstBtn.setEnabled(true);
            preBtn.setEnabled(true);
        }
    }catch(SQLException el){
        JOptionPane.showMessageDialog(mainJframe,"移动游标出错");
    }
}
```

17.3.11 “最后一条”按钮事件响应代码

当用户单击“最后一条”按钮时，它执行的操作和“第一条”相反，不再赘述。

```
protected void doMoveLast(){
    if (curRow>=recordCnt){
        nextBtn.setEnabled(false);
        lastBtn.setEnabled(false);
        curRow=1;
        return ;
    }
    try{
        if (rs.last()){ //将游标移动到最后
            showDate();
            curRow=recordCnt;
            nextBtn.setEnabled(false);
            lastBtn.setEnabled(false);
            firstBtn.setEnabled(true);
            preBtn.setEnabled(true);
        }
    }catch(SQLException el){
    }
```

```

        JOptionPane.showMessageDialog(mainJframe, "移动游标出错");
    }
}

```

17.3.12 “增加记录”按钮事件响应代码

当用户单击“增加记录”按钮时，程序应当转入编辑状态，允许用户编辑一条空记录，但此时并不真的将记录保存到数据库中。只有当用户单击保存按钮时，才将此记录保存到数据库中。因此，也不需要修改和游标有关的任何变量，只要修改记录状态就可以。

为了让用户在成批输入记录时方便，应当允许用户多次按下此按钮，在自动保存上次的记录的同时，再生成一条空记录让用户编辑。

程序应该做以下几件事：

- ☐ 切换编辑状态；
- ☐ 保存输入的数据；
- ☐ 修改按钮状态；
- ☐ 设置文本框状态。

```

protected void doAdd(){
    if (recordState==onlyRead){    //原先是浏览状态，现在切换到编辑状态
        firstBtn.setEnabled(false);
        preBtn.setEnabled(false);
        nextBtn.setEnabled(false);
        lastBtn.setEnabled(false);
        addBtn.setEnabled(true);
        editBtn.setEnabled(false);
        delBtn.setEnabled(false);
        cancelBtn.setEnabled(true);
        saveBtn.setEnabled(true);
        recordState=adding;
        setTextState(true);        //将各个 Text 置为可写
        setTextEmpty();            //将各个 Text 置为空，准备编辑记录
    }else{                        //原先就是编辑状态
        if (doSave(false))        //先保存上次增加的记录
            setTextEmpty();        //如果保存成功，准备增加下一条记录
    }
}

```

17.3.13 “保存记录”按钮事件响应代码

当用户单击“保存记录”按钮时，将调用下面的方法。单击此按钮时，可能处于两种状态：修改记录状态或添加记录状态。该方法需要对这两者进行区别。

另外，本方法可能是由单击了“增加记录”按钮来调用的，或者由用户直接单击了“保存记录”按钮来调用。对于前者，不需要改变当前的编辑状态，对于后者，需要切换回浏览状态。这需要根据调用时的参数来区别。

```

protected boolean doSave(boolean goViewState){
    try{
        if (recordState==amending){           //如果是修改状态
            for(int i=0;i<fieldCnt;i++){
                rs.updateString(i+1, fieldText[i].getText());
                rs.updateRow();                 //更新当前记录
                goViewState=true;              //准备切换回浏览状态
            }else if(recordState==adding){      //这是增加状态
                //将光标移动到准备插入的地方
                rs.moveToInsertRow();
                //下面三步是插入记录必备的
                for(int i=0;i<fieldCnt;i++){
                    rs.updateString(i+1, fieldText[i].getText());
                    rs.insertRow();
                    recordCnt++;                //修改记录条数
                    curRow=recordCnt;           //移动标志
                    rs.last();                  //将光标移动到最后,也就是插入的记录位置
                }
            }catch(SQLException e){
                JOptionPane.showMessageDialog(mainJframe,"保存数据不成功!");
                return false;
            }
        }
        if (goViewState){                      //要切换回浏览状态
            firstBtn.setEnabled(true);
            preBtn.setEnabled(true);
            nextBtn.setEnabled(false);
            lastBtn.setEnabled(false);
            addBtn.setEnabled(true);
            editBtn.setEnabled(true);
            delBtn.setEnabled(true);
            cancelBtn.setEnabled(false);
            saveBtn.setEnabled(false);
            recordState=onlyRead;
            setTextState(false);
        }
        return true;
    }
}

```

17.3.14 “编辑记录”按钮事件响应代码

当用户单击“编辑记录”按钮时,程序需要做以下几件事情:

- ☐ 从浏览状态转到编辑状态,真正更新记录由“保存记录”来完成;
- ☐ 修改按钮的状态;
- ☐ 设置所有文本框为可编辑状态。

```

protected void doEdit(){
    if (0==recordCnt) return ; //如果记录数为零,不可能修改
    //开始设置按钮的状态
    firstBtn.setEnabled(false);
}

```

```

preBtn.setEnabled(false);
nextBtn.setEnabled(false);
lastBtn.setEnabled(false);
addBtn.setEnabled(false);
editBtn.setEnabled(false);
delBtn.setEnabled(false);
cancelBtn.setEnabled(true);
saveBtn.setEnabled(true);
recordState=amending;    //置为修改状态
setTextState(true);
}

```

17.3.15 “取消改变”按钮事件响应代码

当用户单击“取消改变”时，无论是处于增加状态还是修改状态，程序都必须回到浏览状态，同时将游标置为本次修改或增加前的那一条位置，并显示该记录。至于文本框中的数据，则不再需要进行任何处理，因为根本没有写到数据库中去。所以程序需要做以下几件事情：

- ☐ 移动游标位置；
- ☐ 重新显示记录中的数据；
- ☐ 设置按钮状态；
- ☐ 设置所有文本框为不可编辑。

```

protected void doCancel(){
    if (recordCnt==0) return ;
    try{
        rs.absolute(curRow); //移动到原先的记录处
        showDate();
        //设置按钮状态
        if (curRow>1){
            firstBtn.setEnabled(true);
            preBtn.setEnabled(true);
        }
        if (curRow<recordCnt){
            nextBtn.setEnabled(true);
            lastBtn.setEnabled(true);
        }
        addBtn.setEnabled(true);
        editBtn.setEnabled(true);
        delBtn.setEnabled(true);
        cancelBtn.setEnabled(false);
        saveBtn.setEnabled(false);
        recordState=onlyRead;
        setTextState(false);
    }catch(SQLException e){
        JOptionPane.showMessageDialog(mainJframe,"游标移动错误");
    }
}

```

17.3.16 “删除记录”按钮事件响应代码

这里规定删除只能在浏览状态下执行（如果是编辑状态，只要取消就不会保存编辑的记录，相当于删除）。当用户单击“删除记录”按钮时，需要给出一个提示，用户确认后，才能执行删除动作。

但是，删除记录之后，显示哪一条记录则需要仔细考虑，或者是本删除记录的前一条，或者是后一条，但都需要判断是否还有这条记录。下面是详细的代码：

```
protected void doDelete(){
    if(0==recordCnt) return ;           //如果没有记录，什么都不用做
    if (JOptionPane.showConfirmDialog(mainJframe,
                                     "删除后将不可恢复！确定要删除当前记录吗？",
                                     "提示",
                                     JOptionPane.OK_CANCEL_OPTION)
        ==JOptionPane.OK_OPTION){
        try{
            rs.deleteRow();               //删除当前记录
            recordCnt--;
            if(recordCnt>0){               //如果剩余还有记录
                if (curRow>=recordCnt) //如果删除的是最后一条记录
                    curRow=recordCnt;
                //否则的话，curRow的值无需改变，因为后一条记录自动成为了当前记录
                rs.absolute(curRow);
                showDate();
            }else{                         //一条记录都没有了
                curRow=0;
                setTextEmpty();
            }
        }catch(SQLException e){
            JOptionPane.showMessageDialog(mainJframe,"删除数据出错");
        }
    }
}
```

17.3.17 actionPerformed()方法

现在需要实现 actionPerformed()方法，由于所有的按钮都共用同一个监听器，所以需要在该方法中判断事件源，并调用相应的处理方法。

```
public void actionPerformed(ActionEvent e){
    Object obj;
    obj=e.getSource();
    if(obj==firstBtn){
        doMoveFirst();
    } else if(obj==preBtn){
        doMovePrevior();
    }
}
```

```
    } else if(obj==nextBtn){  
        doMoveNext();  
    } else if(obj==lastBtn){  
        doMoveLast();  
    } else if(obj==addBtn){  
        doAdd();  
    } else if(obj==saveBtn){  
        doSave(true);  
    } else if(obj==editBtn){  
        doEdit();  
    } else if(obj==cancelBtn){  
        doCancel();  
    } else if(obj==delBtn){  
        doDelete();  
    }  
}
```

到这里，所有的方法都介绍完毕了。剩下要做的事情是将这些方法都拼装到 AddressList.java 中去。为了节省篇幅，这里不再提供拼装后的程序，请读者自己完成。

17.4 本章小结

本章简要介绍了用 Java 编写数据库应用程序的一般方法，并以一个实际的例子来介绍编写这类程序时的一些技巧以及一些应该注意的问题。当然，这仅仅只是一个入门级的介绍。在数据库方面的应用是 Java 极为重要的一个方面。如果要深入学习，需要阅读更为专业和详细的书籍。另外，在本书的第 19 章 JSP 程序设计以及最后的实例中，还会涉及到数据库编程的一些知识，读者也可进行参考。